References
Simple values and collections

- >>> a = 5
  >>> b = a
  >>> a = a + 1
  >>> b
  5

- >>> a = [1, 2, 3]
  >>> b = a
  >>> a[1] = 5
  >>> b
  [1, 5, 3]

- What’s going on?
Variable contents

- Think of a variable as a “box” that can hold a small amount of information
- In the case of a simple value (number or boolean), the value can fit in the box
- In the case of a larger value, such as a list, what is put in the box is a reference to the value
- What we copy to another variable is what is in the box
Call by value

- When we call a function, the arguments in the call are evaluated, and the values are put into the parameters.
- If the parameters are changed within the function (which isn’t good style), the changes are not put back into the arguments.
  - Besides, the arguments are not necessarily simple variables; they could be literal numbers, or expressions.
- >>> a = 5
  >>> def alter(x):
      x = x + 1
  >>> alter(a)
  >>> print(a)
  5
- We refer to this as *call by value*.
- What is in `a` is a *value*, and that is what is passed to the function.
Call by reference

• When a function is called with something other than a number or boolean, the same rules apply, but something different happens

• >>> b = [1, 2, 3]
  >>> def alter(x):
      x[1] = 99  # changes contents of b
      x = [4, 5, 6]  # does not affect b

  >>> alter(b)
  >>> print(b)
  [1, 99, 3]

• This is *call by reference*

• What is in b is a *reference*, and that is what is passed to the function

• In this example b continues to refer to the same list (the assignment to x does not change that), but the list has been altered
Collections as parameters

- A variable is a “box” that can hold a small amount of information
  - Four bytes of actual data, but there is associated information to describe the type of the data
- Dictionaries and sets are mutable: You can change the values in them
  - Dictionaries and sets are not small, so like lists, a variable whose value appears to be a dictionary or set, is actually a reference to that dictionary or set
  - Python keeps this straight, so you don’t (usually) have to
- **Bottom line:** Dictionaries and sets are like lists; when you pass one into a function,
  - You can change the values in the dictionary or set, but
  - You can’t change it to be a different dictionary or set
Strings as parameters

• Are strings passed to functions by value or by reference?

• ```
  >>> c = "abcde"
  >>> def alter(s):
    s = s + "123"

  >>> d = alter(c)
  >>> print(c)
  abcde
  >>> print(d)
  None
```

• **Answer:** Strings are passed by reference, but since they are immutable, nothing you do in the function will change the original string
Making lists

- You can enter a list directly:
  - ```python
>>> [1, 2, 3]
[1, 2, 3]
  ```

- You can give a sequence to the `list` function:
  - ```python
>>> list(range(1, 4))
[1, 2, 3]
  ```
  - ```python
>>> list({'one', 'two', 'three'})
['two', 'three', 'one']
  ```
  - ```python
>>> list({'one': 1, 'two':2})
['two', 'one']
  ```

- You can “multiply” a list:
  - ```python
>>> ['A'] * 3
['A', 'A', 'A']
  ```
  - ```python
>>> ['a', 'b', 'c'] * 2
['a', 'b', 'c', 'a', 'b', 'c']
  ```
List multiplication gone wrong

• >>> m = [['a', 'b', 'c']] * 3
  >>> m
  [['a', 'b', 'c'], ['a', 'b', 'c'], ['a', 'b', 'c']]

• >>> m[1][2] = '*'
  >>> m
  [['a', 'b', '*'], ['a', 'b', '*'], ['a', 'b', '*']]

• Explanation:
  1. In the first line above, the list contains a reference to a list
  2. The *3 made copies of the reference, not copies of the list itself
  3. This gets assigned to m, which has three copies of the same reference
  4. The second assignment above changes the one referenced list
It’s all about memory

- Variables always hold *small* values--integers, floats, booleans, and references
  - Small values are cheap to copy and pass around
- All other kinds of values are larger (some of which can grow and shrink, like lists), and these larger values are kept in a special part of memory called the *heap*
  - Larger values are expensive (in both time and memory) to copy, so it doesn’t happen automatically
- Two ways to copy:
  - A *shallow copy* of an object makes a copy of an object that includes all the small values (including references) in the original object
  - A *deep copy* of an object makes a copy of that object that includes deep copies of all the referenced objects in the original object
List comprehensions I

- A **list comprehension** is special syntax for generating (creating) a list
- Simple syntax: \([expression \ for \ index \ in \ seq]\)
  - \(seq\) can be a list or an expression that produces an iterator or list
- Examples:
  - >>> \([0 \ for \ i \ in \ range(1, 4)]\)
    
    \([0, 0, 0]\)
  - >>> \([i \ * \ i \ for \ i \ in \ [1, 2, 3, 4]]\)
    
    \([1, 4, 9, 16]\)
List comprehensions II

• You can have multiple `for` expressions:

  • `[(i, j) for i in range(1, 3) for j in range(10, 13)]`
    `[(1, 10), (1, 11), (1, 12), (2, 10), (2, 11), (2, 12)]`

• You can have `if` expressions:

  • `[[i, j] for i in range(1, 30) for j in range(1, 10) if j * j == i]`
    `[[1, 1], [4, 2], [9, 3], [16, 4], [25, 5]]`

• And, of course, you can have multiple `for`s and `if`s:

  • `[[10 * i + j for j in range(2)] for i in range(3)]`
    `[[0, 1], [10, 11], [20, 21]]`

• The *expression* part can itself be a list comprehension:

  • `[[10 * i + j for j in range(2)] for i in range(3)]`
    `[[0, 1], [10, 11], [20, 21]]`
**copy and deepcopy**

- >>> import copy
  >>> m = [1, [2, 3], 4]
  >>> m2 = copy.copy(m)
  >>> m3 = copy.deepcopy(m)
  >>> m[1][1] = 99  # change to a sublist
  >>> m.append(5)  # change to top level

- >>> m
  [1, [2, 99], 4, 5]

- >>> m2
  [1, [2, 99], 4]

- >>> m3
  [1, [2, 3], 4]
Slicing

- Slicing (with the \([i:j]\) notation) produces *shallow* copies
- Using \([:\]) copies the entire list

```python
>>> m = [0, 1, 2, 3, 4, 5]
>>> m2 = m[2:5]
>>> m3 = m[:]
>>> m[3] = 99

>>> m
[0, 1, 2, 99, 4, 5]

>>> m2
[2, 3, 4]

>>> m3
[0, 1, 2, 3, 4, 5]
```
The End